JAVAFO PAIRING ENGINE ADVANCED USER MANUAL

Introduction

JaVaFo is both a stand-alone program (provided that a java virtual machine exist to execute it) and an archive (.jar) that can be used by a program written in the java programming language.

This manual describes the first of the two possibilities, the one that allows an external program to integrate the JaVaFo Pairing Engine in order to use it to prepare pairings according to the Dutch Swiss System *(FIDE Handbook, section C.04.3.1)*.

How to input the data

Albeit JaVaFo supports many input formats (and other ones could be easily added), the most practical way to input data to JaVaFo is using the TRF(x), where **TRF** is the official FIDE Tournament Report File defined in http://ratings.fide.com/download/fidexchg.txt and the **(x)** stands for some extensions that have to be introduced in that format to make it useful for exchanging data between different programs.

The extensions to the TRF are partly made by adding some new codes (which are alphabetic in order to be completely different by the current numeric codes defined for the format[1]), partly by allowing writing in the TRF something that is not normally found in a TRF, partly by interpreting some data contained in it.

Interpretation extensions

A TRF is normally used only to generate data at the end of the tournament. The first extension is to allow the TRF to be generated also during the tournament. This partial TRF is fed to JaVaFo as its input.

It is important that the following codes be used to record unplayed games in the already played rounds:

`0000 - +`	bye win for the unpaired player in a round with an odd number of players
`0000 - =`	arbitral draw (also named *half point bye*)
`0000 - -`	announced absence (i.e. before the pairing was prepared)
`NNNN w -`	absence at the board (i.e. after the pairing was published) of a player scheduled to play with white against NNNN
`NNNN w +`	absence at the board of NNNN, opponent of a player who was scheduled to play with white
`NNNN b -`	absence at the board of a player scheduled to play with black against NNNN
`NNNN b +`	absence at the board of NNNN, opponent of a player who was scheduled to play with black
`NNNN - -`	absence at the board of a player scheduled to play against NNNN with an unspecified colour
`NNNN - +`	absence at the board of NNNN, opponent of the current player (colours of the two scheduled opponents are unknown)

Note that JaVaFo also accepts blank codes which default to

0000	for the opponent
-	for the colour (i.e. no colour)
-	for the result (i.e. forfeit loss).

Therefore for a round already played "`0000 - -`" and " " (eight blank characters) are equivalent.

It is also important that the field called **Points** in the TRF definition (position 81-84) contains the correct number of points that each player got, because that number is used to infer the scoring point system used (i.e. the classic 1, ½, 0 or another one, like 3, 1, 0) .

Unusual info extensions

The partial TRF contains information regarding the rounds that have already been played. However it says nothing regarding the current round (i.e. the one that should be paired). The most important thing is to tell JaVaFo which players should be paired or, which is the same, which players will not play that round.

If everybody plays in the current round, the partial TRF is enough. Otherwise, if somebody is missing, this information is transmitted by inserting in the proper columns for the current round the result code that is normally used for absent players, i.e.

`0000 - -` which identifies an absent or retired player
`0000 - =` which identifies a player who got a half point bye

Actually there is no strict need to differentiate between the two types of absent players. It is just a suggestion for the sake of clarity. The code "**`0000 - -`**" is enough to support any kind of missing player.

Blank codes are ignored in this instance (for obvious reason: each present player is identified by the absence of a result code).

Extra codes extensions

As said in the introduction regarding input data, some alphabetic codes were added in order to transmit additional information to the pairing engine. Some of them are not essential (and will be shown in a following chapter), but one is: the pairing engine must know the total number of rounds in the tournament in order to know if the current round is the last one.

This is obtained by means of the following new alphabetic code:

`XXR` ***`number`***

where *number* is the number of rounds of the competition.

TRF(x) sample

In the linked file, there is an example of the TRF(x) used to pair the fifth round of the **XX Open Internacional de Gros** which includes everything said above regarding the TRF extensions.

TRF(x) Sample

According to what was discussed before, it should be pretty evident that the players 22, 26 and 43 will not play the fifth round.

How to invoke JaVaFo

As mentioned in the introduction, the goal of this paper is to describe how to invoke JaVaFo as a stand-alone program, or, which is a lot more precise, how to invoke JaVaFo as an executable java archive (jar).

First of all, a Java Virtual Machine (JVM) is needed. The beauty of this is that the operating system (O.S.) in use is not important. JaVaFo can run on any O.S. provided that a Java Virtual Machine exist in it. Be **java** the command to activate it.

Then the pairing engine itself, javafo.jar, is needed. The latest version of the jar can always be downloaded from the following web-site:

http://www.rrweb.org/javafo/current/javafo.jar

You can also find previous versions of javafo.jar replacing current with 1.0, 1.1, 1.2 and so on.

The downloaded jar archive can be stored anywhere in the file system. Be JVF_DIR the pathname of the folder where javafo.jar has been downloaded.

The first test to see if both the JVM and the pairing engine work is to write in a command prompt:

java -ea -jar JVF_DIR\javafo.jar (or)
java -ea -jar JVF_DIR/javafo.jar

depending on the O.S. in use (the first line works for Windows; in the following only the Windows commands are mentioned; also for the full command **java -ea -jar JVF_DIR/javafo.jar** the string **javafo** is used, as if a file named ***javafo.bat*** exists somewhere in a PATH directory and contains the statement:

```
@java -ea -jar JVF_DIR/javafo.jar %*
```

If everything is properly set up, the above command should produce the following output (or something similar, more up-to-date):

```
JaVaFo (rrweb.org/javafo) - Rel. 1.4 (Build 1913)
```

After checking that the pairing engine is ready, you can input a TRF to it. Even this file can be placed anywhere in the file system. Be TRF_DIR the pathname of the folder where the file **trn.trfx** is located *(trn.trfx is just a mnemonic name; dummy.foo is an equally valid name)*.

We should also decide where the pairing engine output should be placed. Be **OUT_DIR\outfile.txt** the pathname of said output file (as above, this file can be called in any way). Be sure that the file does not exist before invoking javafo.jar.

From the same command prompt mentioned above, the following command line is needed to produce the pairings for the current round:

javafo TRF_DIR\trn.trfx -p OUT_DIR\outfile.txt

The meaning of the most useful options will be described in a following chapter. But the above command is the most generic way to invoke the javafo.jar pairing engine. If OUT_DIR\ is the same than TRF_DIR\, then OUT_DIR can be dropped, so that

javafo TRF_DIR\trn.trfx -p outfile.txt

generates outfile.txt in TRF_DIR.

How to read the output of JaVaFo

When invoked as shown in the previous command, the output is the file outfile.txt. The structure of this file is very simple:

[1] the first line reports the number of generated pairs (be P)

[2] from the second to the (P+1)-th line, each line contains a pair for the current round. Such pair is made using the pairing-id(s) of the players, i.e. the id(s) that are defined in the **001** line of the TRF(x). The first element of the pair gets white, the second one black. If the number of players to be paired in the round is odd, one of the pairs is formed by the id of the player that gets the bye, followed by 0.

The TRF sample shown above will produce the following output:

```
25
1 2
3 4
5 6
7 13
11 21
23 12
19 16
17 52
35 18
8 24
9 26
37 10
14 29
45 15
46 20
27 38
34 30
39 31
41 32
42 33
44 48
25 49
47 50
51 36
40 0
```

If something goes wrong the output file is not generated. Something is usually displayed on standard output or standard error, but it may be quite difficult to interpret.

The most common cause for an error is a malformed TRF(x). However, if the TRF(x) is totally correct, than an error must have occurred in the pairing engine. This may occasionally happen, but it is a very unlikely event (although not an impossible one, of course).

Extensions and other options

What was presented in the previous chapters covers the majority of the situations that can happen in a tournament. Sometimes, however, something may happen that requires some extra care.

Ranking id

JaVaFo identifies players with two numbers, the player-id and the positional-id.

The first one is obvious and it is the one that is associated with the **001** record in the TRF(x). The second one is implicit being given by the position of the players in the TRF(x). Albeit this is not mandatory, players are normally inserted into the TRF(x) in accordance with their pairing-id, so that the two sets of data (pairing-id(s) and positional-id(s)) are basically coincident.

They may differ, though, and sometimes for good reasons. For instance, please give a look at the linked file, which is a modified version of the previous sample: the players with a local rating (0 for FIDE) are placed at the end of the list.

Ranked TRF(x) Sample

The positional-id is 1 for Mirzoev, 6 for Lakunza (player-id: 7), 29 for Aizpurua (player-id: 32), 42 for Abalia (player-id: 42), 48 for Gorrochategui (player-id: 6) and so on.

JaVaFo computes the pairings using the pairing-id(s). Beware: JaVaFo uses the pairing-id(s), not the ratings, as specified by the Dutch rule A.2.b. This is a programming choice that cannot be modified.

However, the calling program is not forced to follow the same logics. If it desires that the rating be prevalent, it has two alternatives:

[a] before calling JaVaFo, redefine the pairing-id(s) in such a way that increasing pairing-id(s) are assigned to players with decreasing rating (which, by the way, is the most standard situation)

[b] insert players in the TRF(x) in order of rating and tell JaVaFo to use the positional-id(s) (also called ranking-id(s)) instead of the pairing-id(s)

The first choice is the recommended one. However, in order to let the calling program use the second alternative, JaVaFo provides the extension code:

`XXC rank`

The 'C' in XXC stands for configuration. The word **rank** tells JaVaFo to use the positional-id(s) in order to produce the pairings. The output file still contains the pairing-id(s).

First round pairing

Although the rule for pairing the first round is very simple and therefore the calling program can generate it directly, it is recommended to use JaVaFo also to generate the first round.

The only information to pass to the pairing engine is the color of the higher ranked player in the first board. There are three possibilities:

[a] white
[b] black
[c] let JaVaFo make the choice (i.e. random)

The latter choice is the default. Beware that is a semi-random choice: to compute it, JaVaFo uses the hash of some data taken from the TRF(x); this means that repeating the process with the same TRF(x) will give the same result each time. Therefore, in order to use the JaVaFo random choice, the calling program needs doing nothing.

Otherwise, to force the choice [a], the TRF(x) must contain a line

```
XXC white1
```

To force the choice [b], the TRF(x) must contain a line

```
XXC black1
```

Please note that the XXC code is cumulative, so it is followed by all the configuration choices made by the calling program. For instance:

```
XXC rank black1
```

is a valid extension line and combines what was described in the previous and in the current chapters.

Accelerated rounds

The standard way to have accelerated rounds is to assign fictitious points to some players. How to assign such points depends on various methods that have not been yet codified, so this is not a matter of discussion here. However JaVaFo can be informed of the fictitious points that are assigned to each player, using the extension code XXA.

The format for this code is

```
XXA NNNN pp.p pp.p ...
```

Where:

= **XXA** starts at column 1
= **NNNN** (player's id - same as in 001) starts at column 5
= **pp.p** (fictitious points) starts at column 10+5*(r-1), where r is the round which the fictitious points must be added in

It is mandatory to keep the full record of the fictitious points assigned round by round, because this record is used to deduct the floaters history of each player (actually, if pairing for round X, it is enough to maintain the fictitious points history from the rounds from X-3 to the current one - but it seems simpler to keep the full history).

Here is an example from a real tournament where seven accelerated rounds were used:

TRF(x) Acceleration Sample

Forbidden pairs

Sometimes some players are to be prevented from meeting each other. JaVaFo can be directed to fulfill this need by means of the extension code XXP.

The format of this code is:

XXP *list-of-pairing-id(s)*

All the players mentioned in the list will not be paired against each other.
There is no limit on how many times a player can be part of a XXP list. So, for instance, if games between members of two groups of players cannot happen (for instance, let <13, 78, 102> and <68, 111> be these two groups), the following list of XXP extension codes should be generated:

```
XXP 13 68
XXP 13 111
XXP 78 68
XXP 78 111
XXP 102 68
XXP 102 111
```

Long computation

Sometimes the computation of a round may take a very long time.

Let us look, for instance, to the attached TRF(x):

TRF(x) Long Computation Sample

It is a fantasy tournament, of course, but JaVaFo takes sort of an infinite time to compute pairings for the fifth round and it will probably crash. From a practical point of view, it becomes useless.

This kind of problems can be avoided in the following way:

[1] put a limit to the number of permutations that JaVaFo can do in a single bracket (10000 is a good limit; it normally takes less than one minute to pass that limit); this is done by invoking JaVaFo in the following way:

javafo TRF_DIR\trn.trfx -p OUT_DIR\outfile.txt -q 10000

[2] if JaVaFo reaches a number of 10000 permutations in a single bracket, it quits and produces an output file with contains just a 0 (zero, as in zero pairs created).

[3] when the calling program recognizes that such output was generated, it invokes JaVaFo in the following way:

javafo TRF_DIR\trn.trfx -p OUT_DIR\outfile.txt -w

With the option "-w", JaVaFo uses a weighted matching algorithm to compute the pairings; it is slower (sometimes much slower) than the standard one, but it is very useful to solve problems when the normal procedure is sort of stuck.

Check-list

Upon request, JaVaFo can generate a check-list, i.e. a file that summarizes the situation after the pairing, with the following contents (taken from the previous TRF(x) sample):

Check-List Sample

In the check-list, the majority of the fields are pretty intuitive. Some more explanation may be needed for the column **Pref, B5** *and* **B6**.

Pref *reports the preference of the player. The following table explains the symbols and the associated preference:*

W	**B**	Absolute preference (see Dutch rule A.7.a)
(W)	**(B)**	Semi-absolute preference (see A.7.d)

w	**b**	Strong (see A.7.b) or mild (see A.7.c) preference
(w)	**(b)**	Wavering (variable) preference (see A.7.e)
A		No preference (see A.7.f)

B5 *and* **B6** *are references to the Dutch rules B.5 and B.6. They show the kind of floater (up or down) respectively in the last and in the penultimate round.*

In order to produce the check-list, JaVaFo should be invoked using the option **-l,** optionally followed by the name of the file which to put the check-list in. If such name is missing, JaVaFo will produce the file **trn.list**, provided that -l follow the input file name.

For instance, this works:

javafo TRF_DIR\trn.trfx -p OUT_DIR\outfile.txt -l

and this works too:

javafo TRF_DIR\trn.trfx -p OUT_DIR\outfile.txt -l ANY_DIR\outfile.list

As usual, if ANY_DIR is omitted, outfile.list is produced in the TRF_DIR.

Release and build numbers

As mentioned above, the simple command **javafo** will print the release version and build on the standard output.

If an input filename is specified, this information is not output unless the -r option is used.

Pairings Checker

JaVaFo can also be used to check the correctness of a TRF produced by other software: the command line:

javafo TRF_DIR\trn.trfx -c

will produce on standard output something similar to what is shown below, with obvious meaning:

```
trn: Round #1
trn: Round #2
trn: Round #3
trn: Round #4
trn: Round #5
trn: Round #6
trn: Round #7
trn: Round #8
trn: Round #9
  Checker pairings        Tournament pairings
     60 -  51                 60 -  43
     56 -  43                 56 -  51

trn: Round #10
trn: Round #11
```

Using also the **option -w**, the checker will perform its action using the weighted matching algorithm.

Random Tournament Generator (RTG)

In order to help an external pairing-checker, JaVaFo can generate random or quasi-random tournaments against which the external pairing-checker can be tested.

The command line (in its simplest form):

javafo -g -o trn.trf

will generate in the current folder (any pathname like ***TEST_DIR\trn.trf*** can be specified, though) a file **trn.trf**, which is a TRF of a random tournament, with a random number of players (usually between 15 and 215), a random number of rounds (usually between 5 and 15) and game results that depend on the rating difference between the involved players, applying a formula that, in the long run, will distribute points based on the standard FIDE-rating curve.

In the same command line, some of the options already described can be added:

-w	use the weighted matching algorithm
q [*number*]	do not generate a tournament if the number of permutation in any bracket is higher than *number* (or 10000, if no *number* is specified)

The most important utilization of this feature is to generate thousands of tournaments and then test them with the appropriate checker. Therefore it is advisable to use (on Windows) a statement like that:

@for /L %p IN (1000,1,1999) do @javafo -g -o test%p.trf -w

which will generate exactly 1000 random tournaments in the current folder.

The previously mentioned randomness in the generated files can be reduced in two **alternative** ways, using either a (RTG) configuration file or a model TRF.

Reducing randomness by way of a (RTG) configuration file

A (RTG) configuration file is a property-file[2] where the following parameters (properties) may be defined (for the explanation of each single parameter, please look at the sample below):

ParameterName	Default (when the parameter is not defined)
PlayersNumber	*A random number between 15 and 215*
RoundsNumber	*A random number between 5 and 15*
DrawPercentage	*A random number between 10 and 50*
ForfeitRate	*A random number between 6 and 30*
RetiredRate	*A random number between 15 and 3225*
HalfPointByteRate	*A random number between 15 and 3225*
HighestRating	*A random number between 2400 and 2800*
LowestRating	*A random number between 1400 and 2300*
PointsForWin	*1.0*
PointsForDraw	*0.5*

An example of a (RTG) configuration file is shown here:

Random Tournament Generator Configuration Sample

In order to being used by the JaVaFo Random Tournament Generator, the (RTG) configuration file must be specified as a parameter to the **-g** option. Therefore, the full command line is:

javafo -g RTG_DIR\rtg.cfg -o TEST_DIR\trn.trf

Reducing randomness by way of a model tournament

A model tournament is a normal input TRF file with meaningful player ratings, which serves as a model in order to define all the parameters mentioned above.

For instance, from the input file seen in the TRF(x) Acceleration Sample, the following values are automatically retrieved:

ParameterName	Value
PlayersNumber	84
RoundsNumber	9
DrawPercentage	26

ForfeitRate	189
RetiredRate	31
HalfPointByeRate	756
PointsForWin	1.0
PointsForDraw	0.5

The parameters *HighestRating* and *LowestRating* are not considered, as the ratings are exactly the same as the ones present in the input model file.

As the model file is a standard input file, the command line to use it is:

javafo MODEL_DIR\model.trf -g -o TEST_DIR\trn.trf

Quick recap

Standard invocations:

```
javafo [-r]
javafo [-r] input-file -c [-w]
javafo [-r] input-file -p [output-file] [-w] [-l [check-list-file]] [-q [number-of-tries]]
javafo [-r] [model-file] -g [config-file] -o trf_file [-w] [-q [number-of-tries]]
```

The square brackets [ ] represent something optional.

Some explanations:

javafo	indicates a file named **javafo.bat** with the following contents: `@java -ea -jar JVF_DIR/javafo.jar %*`
-r	show JaVaFo release and build numbers

input-file	in TRF(x) format
model-file	in TRF format, model file for the random-tournament-generator
config-file	configuration file for the random-tournament-generator

-c	use JaVaFo as a checker
-g	use JaVaFo as a random-tournament-generator; with this option at most one between model-file and config-file may be present

-p *[output-file]*	**output-file**: full or relative pathname where to write the pairing; if missing, the output file will be defaulted to standard output
-w	use the (usually slower) Weighted Matching Algorithm (WMA)
-l *[check-list-file]*	**check-list-file**: absolute or relative pathname where to write the check-list; if missing, the check-list will be defaulted to the input-file directory, with the same basename as the input-file and an extension of **.list**.
-q *[number-of-tries]*	**number-of-tries**: maximum number of meaningful permutations to be taken in a single scoring bracket; if missing, it defaults to 10000 This option usually has no effect when the **-w** option is also present.
-o trf-file	**trf-file:** full or relative pathname where to write the (auto)generated TRF file

[1]
__ The idea of alphabetic codes comes from the author of the TRF, Christian Krause, chairman of the Swiss Pairings Programs FIDE Commission

[2]
__ A property-file is a file where empty lines have no meaning and lines introduced by the symbol **#** contain a comment.
The meaningful lines have the format ***PropertyName*=*PropertyValue***, which assigns to the named property (or parameter) the value specified by *PropertyValue*.